Image Processing on the NYU Ultracomputer

Robert A. Hummel

Courant Institute, N.Y.U.
251 Mercer Street
New York, N.Y. 10012

Ultracomputer Note #72

November, 1984

ABSTRACT

Many intermediate and higher level vision tasks require long range communication of data within an image. As examples, we consider the connected component labeling problem, and the problem of finding convex hulls of planar regions in a pixel array. While these problems do not admit fast parallel implementations on a locally connected grid array of processors, on a shared memory parallel architecture, such as the NYU Ultracomputer, efficient algorithms can be easily devised. We sketch some of the issues in implementing parallel algorithms for these problems on an Ultracomputer.

1. Image Processing Needs Flexibility

If we regard an image as an array of pixel data and image analysis as a problem of interpreting pixels, then systolic grid arrays of processors would seem to be a logical architecture for parallel processing of images. However, many higher level tasks in image processing require analysis of the interrelation of regions. Further, the regions change dynamically with different images. It is then desirable to disconnect regions of pixels, and to assign groups of processors to each region, with possibly different calculations occurring in each region. Image analysis also frequently requires long range communications, in the sense that intensity data a few hundred pixels distant from a region can affect the interpretation of that region. While pyramid structures can facilitate such communication by decomposing an image into a stack of multiple resolution images, bottlenecks are inevitable if inhomogeneous communication is desired between regions of the image.

Consider, for example, the problems of labeling connected components and finding convex hulls of those components. These are essential tasks in many object recognition systems, such as target recognition applications. Candidate regions of the image can be extracted by thresholding an enhanced intensity image, or some other image of local feature values. Each candidate region must

be analyzed separately (although perhaps in parallel), and the separate segments can be defined as connected components of the binary thresholded image. Convex hulls of regions are useful in several ways. An important feature used to describe a region is the number of concavities in a shape, which we can define as the number of components of the convex hull with the region removed. Convex hulls also provide a good way of smoothing segments, especially if the model segment is convex. Even when the latter condition is not true, shape analysis and matching can take place in a hierarchical fashion using the convex hull of a region and convex hulls of concavities. This process of decomposing a shape by examining concavities can continue recursively until the level of resolution is finer than significant features of the boundary of the objects.

Both connected component analysis and convex hull formation are global problems, requiring long range communication. Two pixels can be in the same component, and this fact might be the result of a long, non-local, circuitous path. In forming the convex hull, a shallow concavity might be filled in by joining two very distant boundary points.

Using a parallel array of locally connected processors, one for each pixel site, connected components and approximate convex hulls can be found by iterative procedures. Connected components are found by iteratively updating labels within regions to agree with the maximum label among neighboring within-region pixels. If each pixel begins with a unique label, this procedure will terminate after a number of iterations not greater that the maximum shortest path length N joining boundary points of any region. Approximate convex hulls can be formed by examining the neighborhood of each non-interior point adjacent to a region boundary. If the nearby boundary is concave, then the non-interior point should be changed to a region point. However, the procedure fails for shallow concavities, and requires that on each iteration a five by five or larger neighborhood be examined at every site. Once again, a large number of iterations are needed.

In both of these algorithms, most processors do no useful work on most iterations. Thus these algorithms waste most of the potential power of the parallel array. The degree of parallelism is so small that it seems unreasonable to devote so much hardware to such simple problems. As is well known, it is possible to apply a sequential algorithm for connected component analysis that is essentially constant time per pixel point. Indeed, this is the approach used in existing target recognition hardware, implemented in a pipeline approach. Sequential convex hull algorithms using a divide and conquer approach, (see, e.g., [1]), exhibit O(MlgM) performance, where M is the number of boundary points.

These algorithms admit effective parallelization, however, but not in systolic grid arrays. For connected component analysis, there are several possibilities. First, the sequential approach, which involves a raster scan of the image, can be broken up into separate component problems in horizontal strips of the image.

After components within each strip are labeled, the interface regions are examined to build an equivalence table of labels that comprise single connected components, due to their connection across interfaces. In a many processor environment, the image can be broken up into smaller blocks. Each processor must be able to access the equivalence table, which is most likely structured as a UNION-FIND forest. A reasonable size for the table will grow as more processors are used. Further, analysis on the interface regions must either be done sequentially, or a queue protocol must be established.

If there are as many processors as interior region pixel points, then another parallel algorithm is available. Rather than having each pixel access its neighbor, we can have each pixel store a pointer to another pixel in the same region having a higher sequential order. On each iteration, each pixel looks at the pixel identified by its pointer, and changes the current pointer to its pointer's pointer. Pointers leapfrog, so that only O(lgM) steps are needed. The algorithm just sketched, however, needs considerable enhancement to work correctly. Pointers can get stuck in local maxima, and so a "hooking" stage is needed on each iteration whereby stuck pointers can hook to larger pixel locations if pointed to by some pixel with a neighbor which points to the higher order pixel. All this can be accomplished in O(lgM) time in a parallel shared memory array, providing concurrent reads and concurrent writes are allowed, and all processors can assess all memory locations. See [2] for details.

For convex hulls, the standard sequential algorithm can be made parallel, so that given M or more processors, $O(lg^2(M))$ time is needed, (where M is still the number of boundary points). In a processor rich environment, the time can be reduced somewhat, or multiple convex hull problems can be solved simultaneously. The algorithm, described in detail in [3], works by first sorting boundary points in the x-dimension and assigning vertical strips of points to individual processors. Each processor forms a convex hull of its points. Convex hulls are merged pairwise. Groups of processors are assigned to each merge operation, in order to find a top and bottom connecting segment for each pair of partial convex hulls. To find each connecting segment, a binary search is conducted in one of the two partial hulls, while an exhaustive parallel search is conducted in the other hull (in constant time) for each iteration of the binary search. For the top connecting segment, for example, a pair of vertices are sought so that successive and previous vertices in each hull lie below the connecting segment. This idea for a parallel search for connecting segments is described in [4].

The algorithms sketched above exhibit efficient parallelism due to the flexibility of processor assignment. In all cases, each processor must be able to access any pixel, and processors are coupled so as to operate in a coordinated fashion. We have concentrated on the connected component and convex hull problems due to their common inclusion in vision systems, even in relatively low-level processing. These are early stages of a vision system which are best

implemented by a shared memory multiprocessor architecture, due to the need for long range communications. The alternatives are to forego any parallelism for these tasks, or to devote an inordinately large amount of parallel resources to a locally connected grid array. The former option carries a large time (or latency) penalty, whereas the latter incurs weight, power, processor resource, and even time penalties.

Higher level image processing stages require even greater flexibility in processor assignments. For example, shape analysis and recognition of each segment can be assigned to separate processors. Furthermore, different features might be extracted at different candidate objects, making independent instruction streams at each processor desirable. Relaxation labeling, stochastic relaxation, and constraint propagation methods for interpreting syntactic components of complicated objects or configurations of objects are amenable to parallel processing architectures. Such methods can be applied on grid arrays as long as the objects to be interpreted are individual pixels. Since the regions of interest are frequently more complicated, dynamic allocation of processing elements is needed for higher-level vision processing. Finally, configuration analysis, and context dependencies, as, for example, in searching for a string of vehicles along an observed road in an image, are issues that are best addressed by either a single sequential processor, or a multiple set of processors each able to access the entire image, each able to communicate or coordinate with all other processors, and each able to concentrate on a dynamically selectable subset of the image.

Although the concept of a shared memory multiprocessor parallel architecture seems simple enough, many issues of coordination and conflict resolution must be addressed. For example, concurrent reads, concurrent writes, and other conflicts are generally resolved by reference to the "paracomputer" model [5], with the "serialization principle": that concurrent conflicting requests be satisfied as though the requests are issued in some arbitrary sequential order. However, the implementation of this principle should not, in fact, invoke serial sections which effectively destroy the parallelism.

In the next section, we briefly outline the architecture of the "NYU Ultracomputer," as it might be configured for image processing and analysis. This architecture provides an approximation to the "paracomputer" concept, and gives a method for configuring the connection network.

2. An NYU Ultracomputer

The architecture of the NYU Ultracomputer is described in detail in [6]. The architecture supports an intermediate number of processor elements (PEs), (any number between 2, and, reasonably, 4096), sharing a large (hundreds of megabytes of) memory, partitioned into memory modules (MMs). The machine admits concurrent reads, concurrent writes, and is augmented by a "fetch-and-add" instruction. When a PE issues a fetch-and-add to a memory location m with a specified increment v, the result is that the processor is eventually informed of

the previous content of memory location m, and the content of that location is subsequently incremented by v. Concurrent fetch-and-adds are handled according to the serialization principle, i.e., as though the requests are issued serially in some arbitrary order. The NYU Ultracomputer is able to handle this task in a highly parallel fashion, incurring no penalty for many concurrent fetch-and-adds.

A plausible and useful configuration of an NYU Ultracomputer for image processing would contain 512 PEs, accessing 512 MMs. The MMs are regarded as an image array, with each MM containing 512 pixels from a 512 by 512 pixel array. Conceptually, we could assign one row of the image to each MM, although practically it may make better sense to use a less regular assignment. Each MM would be many kilobytes deep. In this way, the MMs can store several images, including images of feature values, and have room left over for buffers and tables. An address into one of the MMs consists of two 9 bit numbers (i, j), specifying a pixel row and column coordinate, followed by a description of the bits to be accessed within the pixel, represented as an offset. If each MM is used to store a row of the image, then the first nine bits (i) give the MM number, while the column j and pixel description are used to calculate an address within the MM. If the pixels are assigned to MMs in a more complicated fashion, then the pair (i, j) is used to calculate the MM number, by a hash function or mapping, and also is used in the calculation of the offset within the MM.

The PEs are connected to MMs through an Omega - Network, [7], consisting of nine stages of switches. Each stage will contain 256 switches; each switch has two inputs and two outputs, routing requests from PEs to MMs. A request issued to an MM is routed through the network by using the bit representation of the MM number. In the kth stage of switches, the kth bit (counting from the msb) is used to specify which of the two output lines of the current switch to which the request should be sent. The request emanates from this port, and is connected to the proper switch in the $(k+1)$st stage (or the appropriate MM in the last stage). For any specified address (i, j), there is a unique path from each PE leading to the MM containing the (i, j) pixel. Further, while the request passes through the network, the return address can be built up by noting successive input port numbers on each switch in the path. A return address is needed for each load request, and also for fetch-and-add instructions. Data is passed both forward and backward through the network, so that, for example, a load request will require 18 clock cycles to fulfill. Requests are pipelined through the network, however, so that requests to memory incur a 9 or 18 cycle latency. Further, read-only variables can be cached in local PE memory.

Concurrent requests to a single MM will generally meet at some intermediate stage in the network. The switches are designed to resolve such conflicts before concurrent requests are passed on to the next stage. In particular, the arithmetic for concurrent fetch-and-add instructions is carried out mostly within the network. For many fetch-and-adds to the same address, the arithmetic will be

distributed throughout the network. Moreover, each switch stores information about concurrent requests involving fetches, so that all requesting PEs can be serviced by a single access to the appropriate MM, by having the returned data bifurcate while passing backwards through the network at those switches where requests were combined.

Note also that although the path from a PE to an MM is unique, paths can intersect at switches. Thus requests to separate MMs issued by different PEs might meet simultaneously at a single switch. Since the switch can service only one request through each output port during a given cycle, the switch will have to queue requests for each output port. Simulation studies have suggested that queues of lengths of eight or so requests will generally never fill.

If many PEs attempt to concurrently access different data within one MM, then a bottleneck will occur. The requests cannot be combined, since they access different data, yet the MM to network interface has only a limited bandwidth. In essence, the network will effectively serialize the requests. This could destroy the advantages of the parallelism, and queues might fill in severe circumstances. For example, if all PEs are assigned to work on one row of the image data, and that row is stored within one MM, then problems will occur. There are several ways out of this dilemma. The approach that we take below is to simply avoid assigning all PEs to one row, and instead attempt to distribute the PEs among as many rows as possible. Another possible approach would be to hash the pixels among the MMs, so that each MM is assigned a random set of pixels from the image, and many concurrent requests to different pixels within one MM is unlikely. Finally, if the image array is small, and hardware size is no object, then it is possible to build an interconnect net which fans out, connecting, say, 64 PEs to a 64 by 64 (4096) array of MMs. In this configuration, each MM would store data associated with one pixel. The switches would have 2 input lines and 4 output lines, with 32 switches in the first stage, and 1024 switches in the 6th and last stage. Bottlenecks won't occur unless many PEs try to access different data items at one pixel. However, the disadvantage of this approach is that a great deal of hardware is devoted to the interconnect network, with the bandwidth limited by the narrowest point in the net (namely, the first stage).

The fetch-and-add instruction allows many operating system functions to be distributed throughout the PEs, so that there is little need for a master PE, [8]. For example, suppose that we must assign each PE uniquely to exactly one of 512 segments in the image. Each PE should issue a fetch-and-add with increment 1 to a single memory location initially containing the integer 0. Each PE will be returned a unique integer in the range 0 to 511, which denotes the segment index to which that PE is assigned.

The parallel computer may be programmed as 512 independent processors, each able to access all image memory. The machine is of the MIMD class, since multiple instruction streams can occur - a different instruction at each PE. Although a variety of PE to MM instructions are provided, they all fall into the

categories of load, store, and fetch-and-add.

Using this configuration of the NYU Ultracomputer architecture, many of the image processing tasks mentioned in Section 1 are straightforward. We also wish to note that for many tasks, the 18 cycle latency (which can grow longer when collisions occur in the network, invoking output port queues) is inconsequential, since most requests to memory will be a stream of memory transfers between MMs and PE local memory, pipelined through the network. For example, each processor can be assigned to one image row for connected component analysis. After the row of image data is cached in a processor's local memory, concurrently processed, and written back to memory, then each processor is reassigned to an interface between rows. UNION-FIND requests can be written in coded form to a designated area of memory, and available processors assigned to accessing that area to service the requests, building up an equivalence table. Once the table is completed, it can be loaded into the local memory of each PE, since it is now read-only data, and applied as a point look-up-table operation to pixel label information, each processor handling one row of the image in pipeline fashion. Assuming a clock cycle time of 100 ns (which can probably be improved considerably), the whole operation should take less than 1 msec. By comparison, a sequential pipelined approach would generally incur a two-frame time (67 msec) latency period, whereas a comparable processing time can be achieved on a systolic grid array only if a quarter meg of processors are equipped with a special 1 - 10 microsecond operation for finding the max of labels among neighboring activated processors.

For the convex hull problem, the shared memory multiprocessor environment is essential if exact convex hulls are to be computed. Further, since no single component is likely to have as many as 512 boundary points ($M <= 512$), the processors can be divided up among one or more convex hull problems, one processor to each boundary vertex, yielding very efficient use of the parallel processors. The search for top and bottom connecting lines between partial convex hulls proceeds by distributing the tasks of checking the viability of potential segments to separate processors. The tentative vertex in the one partial convex hull, sought by a binary search, is determined by a single PE, which communicates that information to a set of other processors, each assigned to check a vertex in the other convex hull, by writing the first vertex's location, and other relevant information, in some fixed memory location. In this case, network latencies are more significant, but tolerable due to the amount of parallelism involved in the coordinate location calculations needed to compute convex hull merges. Suppose, for example, that 512 processors are put to work on a half dozen convex hull problems, where each object has less that 100 boundary vertices. Then roughly 6 or 7 hull merging periods are needed, and each hull merge will involve fewer than 6 candidate vertices in the binary search, with perhaps 200 clock cycles needed to determine the vertex in the second partial hull to link with the candidate vertex in the first hull. The 200 cycles includes network delays caused by the need for all processors to access relevant coordinate

information about the associated candidate vertex. In any case, the processing time is well under 1 millisecond.

The true power of the architecture is evidenced by the fact that no hardware is dedicated to these particular algorithms. The same set of processors can then proceed to work on image analysis tasks and higher level constructs, accessing image memory and/or descriptive information about regions stored in buffers in the MMs. Processors can share description information due to the shared memory construction. Indeed, the idea of having one or more high speed processors able to access all image memory and other storage is extremely uncontroversial. Our point is that if the number of such processors is sufficiently high (like 512), then tasks such as connected component analysis and convex hull formation can be solved efficiently. However, when that many processors are involved, a hardware designer will face serious questions of conflict resolution and interconnection. If the designer is not careful, much of the power of the parallelism will be lost. The NYU Ultracomputer provides a model for configuring the interconnections, and implementing concurrent accesses to memory.

The same architecture can also be used for low level image processing operations. While most local operations achieve higher throughputs when hundreds of thousands of simple locally connected processors are involved, the 512 processor architecture outlined here is certainly very powerful. Processing times are dominated by the long range connection algorithms anyway, so that inefficiencies in local operations are not likely to significantly hinder processing speeds. Finally, if a grid array of processors is deemed desirable, they can be built into the memory modules configured as a relatively deep image array, providing the pixels are assigned to MMs in a structured way. As we noted earlier, it may be useful to hash pixels into the MMs, in which case an SIMD grid array within the MMs is not possible. However, it would be reasonable to have such a grid array within the MMs for early processing, and at some intermediate processing stage read the data into the PEs and hash back into the MMs, at which point the local connections in the grid array would no longer have any use.

Special VLSI chips can be (in fact, have been) designed for the switches [9], so that the interconnection network can be constructed using very few distinct chip types. Likewise, memory modules are all identical, and PEs and the interface to the network is highly regular. The most difficult part of the construction of an NYU Ultracomputer consists in wiring the Omega-network. The wiring cannot be placed on planar silicon, since there are many crossovers needed. Packaging and wiring considerations have been addressed for a 4096 PE - 4096 MM NYU Ultracomputer [10], and lead to an enclosure suitable for fixed placement, but not recommended for mobile operation. Considerable savings can be incurred by increasing the number of input and output ports on each switch. For example, a 256 PE to 256 MM image array, using 4 port in, 4 port out switches, would require only 4 stages of switches, with a total of 256 switches in

the network.

REFERENCES

(1) Preparata, F.P., "Convex Hulls of Finite Sets of Points in Two and Three Dimensions," *CACM 20*, 1977, p. 87.

(2) Shiloach and Vishkin, "An O(logN) Parallel Connectivity Algorithm," *J. of Algorithms 3*, 1982, p. 57.

(3) Nath, Maheshwari, and Bhatt, "Parallel Algorithms for the Convex Hull Algorithm in Two Dimensions," in *CONPAR 81*, *Proceedings*, Goos and Hartman (eds.), Lecture Notes in Computer Science 111, Springer-Verlag.

(4) Preparata, F.P., "An Optimal Real-Time Algorithm for Planar Convex Hulls," *CACM 22*, 1979, p. 402.

(5) Schwartz, J. T., "Ultracomputers," *ACM TOPLAS*, 1980, p. 484.

(6) Gottlieb, Grishman, Kruskal, McAuliffe, Rudolph, and Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. on Computers 32*, 1983, p. 175.

(7) Lawrie, D., "Access and alignment of data in an array processor," *IEEE Trans. Computers 24*, 1975, p. 1145.

(8) Gottlieb, A., B. Lubachevsky, and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," *ACM TOPLAS*, Jan., 1983.

(9) Snir, M. and J. Solworth, "The Ultraswitch -- A VLSI Network for Parallel Processing," Ultracomputer Note 39, Courant Institute, New York University, 251 Mercer, NY, NY 10012.

(10) Bianchini, R. and R. Bianchini, Jr., "Wireability of the NYU Ultracomputer," Ultracomputer Note 43, Courant Institute, New York University.